Simulating evolutionary agent communities with OOCSMP

Manuel Alfonseca[1]
[1]Dept. Ingeniería Informática,
Universidad Autónoma de Madrid
Ctra. De Colmenar, km. 15, 28049 Madrid, Spain
Tlf.: +34 91 348 22 78
Manuel.Alfonseca@ii.uam.es

Juan de Lara[1,2]
[2]School of Computer Science
McGill University
Montréal, Québec, Canada
Tlf.: + 34 91 348 22 77
Juan.Lara@ii.uam.es

ABSTRACT
This paper describes some extensions added to the continuous simulation language OOCSMP to perform agent-oriented simulation. The extensions are tested by simulating the evolution of a colony of virtual ants (vants). In this simulation, each vant is modelled as an agent and is assigned a set of genes that control some aspects of its behaviour, such as its velocity, memory, communication abilities, scepticism, etc. Some emergent properties of the swarm of vants have been observed.

Categories and Subject Descriptors
I.6.2 [**Simulation Languages**], I.2.11 [**Distributed Artificial Intelligence**] - Coherence and coordination, Intelligent agents, Languages and structures, Multiagent systems

General Terms
Design, Experimentation, Languages.

Keywords
Agent-based simulation, Swarm Intelligence, Multi-agent languages, Artificial ants, Evolution, OOCSMP.

1. INTRODUCTION
Agent-based simulation is a powerful and natural way to carry out complex simulation experiments where many autonomous and interacting entities take part. The key abstraction in this methodology is the autonomous agent. According to [1], an agent is "a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives". Agents interact via discrete events.

Several approaches can be followed when implementing a multi-agent system [2]: Logic-Based Architectures, Belief-Desire-Intention (BDI) Architectures and reactive and layered architectures. [3,4]. This work has taken the latter approach.

Agent-based simulation can be used with different objectives:

- Resolution [5] and optimisation [6] of mathematical problems.
- Study of emergent global behaviour and social interactions [7, 8].
- Study of population tendencies and evolution [9].

One of the most interesting things to study in this kind of systems is emergence [10]. This phenomenon occurs when interactions in a large population of objects at one level give rise to different types of phenomena at another level.

OOCSMP is an object-oriented extension of the old CSMP [11] continuous simulation language, sponsored by IBM in the seventies and the eighties. OOCSMP is specially useful when the system to be modelled is composed of similar components that interact. Other extensions added to OOCSMP make it easy to solve partial differential equations or produce distributed simulations.

This paper presents some new language capabilities to perform agent-oriented simulation. The extensions are tested simulating an agent community similar to a colony of virtual ants (vants). The objective of this simulation is not to model realistic ants, but to experiment with several aspects of communication (vants communicate directly, not by means of pheromones) and evolution (every vant is provided with genes and reproduces sexually, unlike real ants). Interesting emerging behaviour has been observed.

The paper is organised as follows: section 2 gives a quick overview of OOCSMP; section 3 describes the extensions to perform agent-oriented simulation; section 4 presents the basic scenario for the experiments; section 5 shows the main results of the simulations; section 6 summarises with the conclusions and future work.

2. OOCSMP: AN OVERVIEW
The OOCSMP language was designed in 1997 [12] as an object oriented continuous simulation language. A compiler (C-OOL) was built for this language to produce C++ code or Java applets from the simulation models. This approach would simplify the generation of simulation based web courses, because the user does not have to worry about Java or HTML low-level details. In fact, a number of courses have been generated using this language: gravitation, partial differential equations, ecology and basic electronics [13], which are accessible from: *http://www.ii.uam.es/~jlara/investigacion*

The language and the compiler have been designed with an educational focus. If Java is chosen as the object language, a user interface is generated automatically in which the user can answer "what if...?" questions in a "learning by doing" paradigm. When performance is a must, the compiler may be instructed to generate C++, although the user interface in this case is restricted.

Although it was conceived as a continuous language, OOCSMP has features that allow including a certain degree of discrete simulation in the models: it is possible to handle discrete events by means of blocks *INSW* and *FCNSW*. When a discrete event takes place, which affects a variable appearing in an expression that should be integrated, the corresponding integrator is automatically reset to process the discontinuity.

3. EXTENDING OOCSMP FOR AGENT-ORIENTED SIMULATION

OOCSMP is very useful if the model can be expressed as a collection of similar entities that interact, because the entities can be modelled as collections of objects, and the interactions as method invocations. Several extensions have been added to OOCSMP to perform agent-oriented simulation:

- An agent can be modelled as an OOCSMP object. OOCSMP classes represent types of agents. Each class defines the agent behaviour (by means of the available methods) and its state variables (by means of attributes).
- Multiple object constructor invocation is supported. A single instruction can declare several 'unnamed' objects.
- Objects can be added to, or deleted from a collection, using the overloaded operators '+=' and "-=".
- Objects can be eliminated from the simulation using the DELETE operator. The compiler makes a static analysis of the model to optimize the handling of the "*dead*" objects.
- The new SELF keyword refers to the addressed object. Among other things, this permits the object to add or delete itself from collections, or eliminate itself from the simulation.
- A new output form represents the position and the state of the agents. The graphical representation of an agent can have different shapes, such as rectangles, triangles, circles, etc. The state can be represented as the colour and/or the size of that shape.
- Instructions to repeat (and change) experiments, and collect statistical data.
- According to [14] a point-to-point message passing mechanism would restrict the power of a multi-agent system. For that reason, multicast and broadcast message-passing mechanisms have been implemented in OOCSMP. In this way, methods can be invoked on objects (point-to-point), classes (broadcast) or collections of objects (multicast). In the two last cases, an implicit iteration is generated, which invokes the method on each object of the class/collection. The order in which the elements of the class/collection are accessed can be sequential (first to last or last to first), random or specified by the user in a vector. The syntax for method invocation on classes or collections is shown in table 1.

  According to [4], random access to the elements of the collection can be necessary in agent-based simulation to avoid artefacts, i.e. phenomena that arise due to accidentally imposed inter-agent correlation.

Table 1: Syntax for method invocation on classes or collections of objects

Syntax	Meaning
`<collection>.<method>(<args>)`	Invocation of the method on all the elements in sequence.
`<collection>[<].<method> (<args>)`	Invocation of the method on all the elements in reverse order.
`<collection>[?].<method> (<args>)`	Invocation of the method on all the elements in random order.
`<collection>[<vector>].<method> (<args>)`	Invocation of the method in the order given by the elements of the vector.
`<collection>[<scalar>].<method> (<args>)`	Invocation of the method on the element given by the scalar expression in square brackets.

- If a method returns a scalar value, and is invoked on a class or a collection of objects, the global result is a vector; each element of the vector is the result of applying the method to each object in the class/collection. If the method returns a vector, the global result is a matrix.
- If one of the arguments of a method is an object, the method may be invoked replacing that argument by a class name or a collection of objects. In this case, an implicit iteration is generated, and the method will be invoked for each element in the class/collection. The order of access to the elements in the class/collection can be modified in a way similar to table 1. If both the target of the method and its argument are collections of objects or classes, a double iteration is generated, and the method is invoked for every object in the target and each object in the argument, except when both are the same object. This situation is useful, for example, when agents want to communicate with the other agents in the same collection, excluding themselves. A similar situation arises when one of the method arguments is a scalar and is invoked with a vector.

4. SIMULATION OF AN EVOLUTIONARY VIRTUAL ANT COLONY

The extended OOCSMP has been used to model an artificial foraging vant community. The aim was not to be realistic, but to experiment on knowledge propagation between agents. Thus, vants communicate directly with other vants when they are near, rather than by dropping pheromones.

Vants live in a two-dimensional grid of size 50x50. Their objective is to find food. When they are successful, they eat a portion (which extends their life span), and take another portion to the nest. This may be repeated until the food is depleted. Several locations with food may exist at the same time. When a

vant arrives at the nest, it rests there for some time. When two agents meet outside the nest, they may exchange their knowledge about the food position. If a vant does not find food during a certain period of time, it returns to the nest.

Figure 1 shows a state transition diagram (STD) [15] for the vant. STDs are used broadly to express the dynamic behaviour of software systems and are a natural way to express agents' behaviour. Other ways to express agents behaviour can be found in [2, 16]. Observe that some transitions depend on non-deterministic conditions, i.e. whether a certain parameter is greater or smaller than a random number.

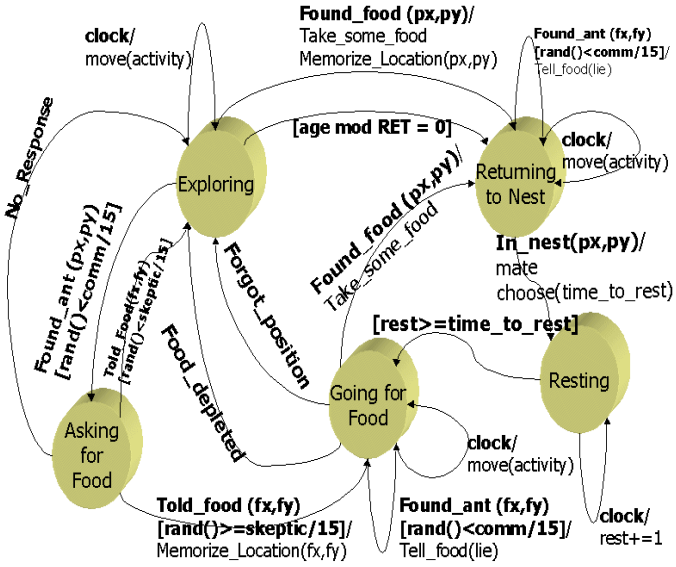


Figure1: STD describing the behaviour of a vant.

Vants can be in one of five states:

- Exploring randomly, when the agents don't know the location of any food source.
- Returning to the nest, when the vant has found food and at predefined intervals.
- Resting in the nest for a brief, random time.
- Going to fetch food, when the agent knows the food location.
- Exchanging information with another vant, if they meet and one of them doesn't know any food position and both decide to talk (this is controlled by the communicative attribute).

Our vants have several parameters that control their behaviour:

- Activity: It controls the speed of each vant. It has four possible values.
- Communicative (*comm* in the picture): It is used to decide if the agent will communicate with another agent when they meet.
- Scepticism (sceptic in the picture): This parameter controls the credulity of the agent. At one end, the agent always believes the information about the food location received from the other agent. At the other, it never trusts that information.
- Lie: This parameter controls the degree to which agents lie when they inform the others of the food position. It is a number between 0 and 3, with the following meanings:

  0: The agent always tells the truth.

  1: The agent communicates the approximate position.

  2: The agent provides a random position.

  3: The agent sends its partners in the opposite direction.
- Memory: This parameter doesn't appear explicitly in the picture, but it controls the probability that an agent forgets the food position it knew about.

The five parameters are encoded in binary and concatenated, making a genotype. When two agents meet at the nest, they can reproduce if there's enough food in the nest. In each reproduction, two new agents are created, with 'genetic' information resulting from their parents genomes after the operations of mutation and (uniform) crossing-over have been applied to them, a typical procedure in genetic algorithms [17]. A scheme of the reproduction is shown in figure 2.

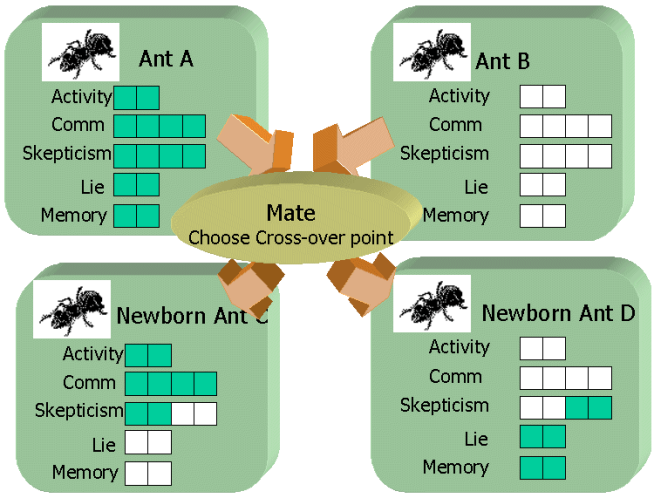


Figure 2: Vant´s reproduction

Reproduction is only allowed in the nest, and it only happens if there is a minimum amount of food, because the new-born agents are assumed to need some food to grow.

Other attributes are needed to implement the agent's state, such as its current position, the position of its nest and its maximum age. The last attribute is set initially for each agent as a random number with a gaussian probability (average 250, standard deviation 50). This attribute decreases after each time step and increases when the agent gets food. RET is a global parameter (with the same value for all the vants) that controls the intervals for the exploring vants to return to the nest.

In this implementation, each vant is represented as an OOCSMP object of class AGENT. There's also a class (NEST) that contains a collection with all the vants belonging to the nest, and collects statistical data. Another class (TERRITORY) takes care of food sources, manages the amount of food in each, and generates randomly a new source when one is completely depleted. Listing 1 shows the code for the NEST class.

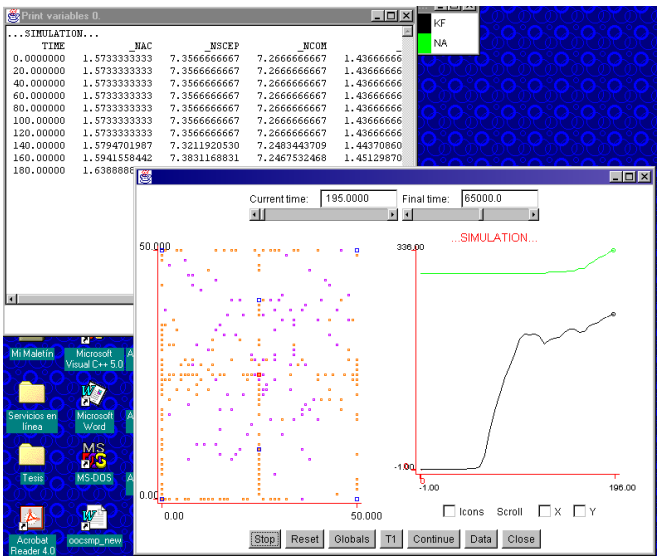
```
[1]    INCLUDE "Territory.csm"
[2]    INCLUDE "Agent.csm"
[3]    CLASS NEST{
[4]    TERRITORY T1
[5]    DATA NUMAGENTS:= 300, POX:=0, POY:=0, ANTHILL :=
           0
[6]    AGENT LAGENTS := AGENT [NUMAGENTS]( SELF )
[7]    INITIAL
[8]      MAXX := T1.MAXX
[9]      MAXY := T1.MAXY
[10]   ...
[11]   DYNAMIC
[12]    FCNSW(ANTHILL, , ,LAGENTS[?].MATE(LAGENTS[?]))
[13]     LAGENTS[?].STEP()
[14]     LAGENTS[?].COLLIDE(LAGENTS[?])
[15]     KF := 0
[16]     KF += LAGENTS.KNOWFOOD
[17]     NA := 0
[18]     NA += INSW ( LAGENTS.MAXAGE, 0, 1)
[19]     NL     := MEAN (LAGENTS.GETLIE())
[20]     NAC    := MEAN (LAGENTS.GETACTIVITY())
[21]     NSCEP  := MEAN (LAGENTS.GETSCEPTIC())
[22]     NCOM   := MEAN (LAGENTS.GETCOMMUNICATIVE())
[23]     NMEM   := MEAN (LAGENTS.GETMEMO())
[24]   PRINT NA, NAC, NSCEP, NCOM, NL, NMEM}
```

Listing 1: Nest class

Lines 4-7 declare some attributes. The INITIAL section starts at line 8 (executed only once, after the constructor of the object is called). Several auxiliary methods are not shown (line 10). Line 11 begins the declaration of the DYNAMIC section, which executes once per time step. First of all (line 12) it checks if there's enough food in the nest to allow reproduction. If this is the case (discrete event, handled by FCNSW), it iterates on the vant collection, in random order. Line 13 calls the DYNAMIC section for each vant (this is done by the STEP method invocation) in random order. Line 14 processes communication between vants. The following lines collect statistics (gene distribution in the population), which are printed at line 24. Since the PRINT instruction is located inside the NEST class, the statistics of all the objects in class NEST are shown; this is a useful feature that makes scalability easier.

5. SIMULATION RESULTS

Since we need a lot of computing power, we have compiled our examples into C++. Figure 3, however, shows a slower simulation compiled into Java (the user interface has been generated automatically with C-OOL) with only one nest and two graphical output forms:



- At the left, a plot representing the position and state of the vants, nests and food sources.
- To the right, an animated 2-dimensional plot shows the number of vants (the upper line, in green) and the number of vants that know the location of a food position. Agents may lie: some of those that think they know the location of a food source may be wrong.
- The result of the PRINT instruction in listing 1, line 32 is shown in the background window.

Figure 3: A moment in the simulation (Java Interface)

We have found that the average of the Activity parameter in the population grows quickly to its maximum value, because the fastest vants have an evident advantage on the others. The same happens with the Memory parameter, which grows quickly to the maximum (the larger a vant can remember a food position, the better, because, the vant can return several times for food until it is depleted). The other parameters may oscillate, but we have identified two situations for the case with a single nest:

- When food is scarce, agents compete between themselves and liars begin to proliferate. As this parameter goes up, scepticism also grows. The explanation is clear: in this situation it is more advantageous not to trust the others, because if a vant trusts a liar, it can be sent to a completely wrong position. A false information can be propagated quickly among the population: this gives rise to the appearance of rumours. Since rumours are clearly bad for the community, it defends itself by increasing scepticism.
- When there's plenty of food, it's better for vants to co-operate, and liars may disappear quickly. The same happens with sceptic vants. The explanation for this is also clear: if there are few liars, it is much more advantageous to trust. It must be noted that, if there's plenty of food, the number of liars does not decrease always but, if this happens, it only happens when there's plenty of food.

This is an emergent behaviour of the system, which does not include an explicitly programmed correlation between lying and scepticism.

The behaviour of the other parameter (Communicative) is less clear, but it tends to be higher in situations of abundance. In such situations, the Activity and Memory parameters tend to grow more slowly, as there is not such a selective pressure. On the contrary, when food is scarce, these two parameters tend to grow very quickly to their maximum values.

6. CONCLUSIONS

This paper has described some extensions that turn OOCSMP in a good choice for agent-oriented simulation. Some useful features are the facilities to iterate on collections of objects using different schemes, and the possibility to invoke methods on objects, classes and collections of objects. OOCSMP was conceived as a continuous simulation language, thus it is also possible to take advantage of powerful features such as integrals, derivatives, solving partial differential equations, etc.

The compilation scheme adopted, lets the experimenter choose between two alternative situations:

- If performance is needed, compilation into C++ can be better. Additional hand optimisation can be done if needed.
- Compilation into Java is a better option if there is a need to inspect visually the results of the simulation. With the automatically generated Java user interface, parameters can be changed during the simulation execution. Due to Java slower performance, as compared to C++, these experiments usually contain a smaller number of agents.

The language extensions have been tested with the simulation of an evolutionary vant colony, using STD's as a general tool to describe the agents' behaviour. In the simulation, interesting results have been observed, such as a correlation between liar and sceptical agents.

The model will be extended by making the agent's behaviour more complex. They will be given a new gene that controls their 'Aggressivity,' so they will be able to rob or kill agents belonging to different nests. If a nest is too crowded, it will split and generate a new anthill. Another interesting extension would be modelling scents, pheromones and other types of indirect communication.

As OOCSMP has primitives to generate parallel simulations, we will explore possible parallel implementations of the model.

We are planning to enhance the discrete possibilities of our language with event queues, event types, etc. We are also thinking of adding an external API to call C++ or Java functions from OOCSMP, and enhancing the mechanisms of OOCSMP for handling objects and collections of objects. Detailed comparisons between OOCSMP and other simulation languages [3, 4] are also needed.

7. ACKNOWLEDGEMENTS

This paper has been sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project number TEL1999-0181

8. REFERENCES

[1] Jennings, N.R., Sycara, K., Wooldridge, M. "A Roadmap of Agent Research and Development". Autonomous Agents and Multi-Agent Systems, 1, 7-38 (1998). Kluwer Academic Publishers.

[2] Wooldridge, M. "Intelligent Agents". In "Multiagent Systems. A modern approach to Distributed Artificial Intelligence" (Weiss ed.). pp. 27-77, The MIT Press. 1999.

[3] Swarm home page: http://www.swarm.org

[4] Axtell, R. "Why Agents? On the varied motivations for agent computing in the social sciences". Working paper nº17 at Center on Social and Economic Dynamics. Brookings Institution. 2000.

[5] Drogoul, A. "When Ants Play Chess (Or Can Strategies Emerge From Tactical Behaviors?)". LNAI, nº 957, pp. 13-27, Springer-Verlag, Berlin-Heidelberg (1995.).

[6] Dorigo, M., Maniezzo, V. "The Ant System: Optimization by a colony of cooperating agents". IEEE Transactions on Systems, Man, and Cybernetics, Part-B, Vol.26 (1996), No.1, pp. 1-13.

[7] de Lara, J., Alfonseca, M. "Some strategies for the simulation of vocabulary agreement in multi-agent communities". JASSS vol. 3 (2000), no. 4, <http://www.soc.surrey.ac.uk/JASSS/3/4/2.html>

[8] Hraber, P.T., Jones, T., Forrest, S. *"The Ecology of Echo"*. Artificial Life 3:165-190. (1997).

[9] Ophir, S. "Simulating Ideologies". Journal of Artificial Societies and Social Simulation vol. 1 (1998), no. 4, <http://www.soc.surrey.ac.uk/JASSS/1/4/5.html>

[10] Gilbert, N., Troitzsch, K. "Simulation for the Social Scientist". Open University Press. 1999.

[11] IBM Corp.: "Continuous System Modelling Program III (CSMP III) and Graphic Feature (CSMP III Graphic Feature) General Information Manual", IBM Canada, Ontario, GH19-7000, 1972.

[12] Alfonseca, M., Pulido, E., Orosco, R., de Lara, J. "OOCSMP: an object-oriented simulation language". ESS'97, Passau, pp. 44-48. 1997.

[13] Alfonseca, M., de Lara, J., Pulido, E. "Semiautomatic Generation of Web Courses by Means of an Object-Oriented Simulation Language", special issue of "SIMULATION", Web-Based Simulation, Vol 73 (1999), num.1, pp. 5-12.

[14] Fisher, M. "Representing and Executing Agent-Based Systems". In Wooldridge and Jennings (eds), Intelligent Agents. LNCS 890. Springer-Verlag (1995).

[15] Pressman, R.S. "Software Engineering: A Practitioner's Approach". 4th Edition. McGraw Hill, 1997.

[16] Wooldridge, M., Jennings, N.R., Kinny, D. "The Gaia Methodology for Agent-Oriented Analysis and Design". Journal of Autonomous Agents and Multi-Agent Systems 3 (3) 285-312. Kluwer Academic Publishers (2000).

[17] Holland, J.H. "Adaptation in Natural & Artificial Systems." University of Michigan Press (1975).